

UNIVERSITY OF
ILLINOIS LIBRARY
AT URBANA-CHAMPAIGN

~~MATHEMATICS~~



Digitized by the Internet Archive
in 2013

<http://archive.org/details/crossassemblersi886blan>

510.84
IL6r
no. 886
cop. 2

A CROSS-ASSEMBLER/SIMULATOR FOR A MICROPROCESSOR-
BASED INDUSTRIAL CONTROLLER

by

Robert Elmore Blanton

July 1977



DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

The Library of the

NOV 02 1977

University of Illinois
at Urbana-Champaign



UIUCDCS-R-77-886

A CROSS-ASSEMBLER / SIMULATOR
FOR A
MICROPROCESSOR-BASED INDUSTRIAL CONTROLLER

BY

ROBERT ELMORE BLANTON

B.S., Northeastern University, 1975

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1977

Urbana, Illinois

510.84
IL6r
no. 886-893
cop. 2

A CROSS-ASSEMBLER / SIMULATOR FOR A MICROPROCESSOR-BASED
INDUSTRIAL CONTROLLER

Robert Elmore Blanton
Department of Computer Science
University of Illinois at Urbana-Champaign, 1977

Since the advent of the microprocessor, many areas have become increasingly open to computerization; one such area is industrial process control. Throughout the past decade, industrial controllers have been dominated by relay logic. Various methods of upgrading these controllers through the use of solid-state discrete logic controllers or large (mini or maxi) computers have been met with varying degrees of success. Solid-state discrete logic controllers, while being faster than their relay logic predecessors, also inherit one of their major drawbacks - their lack of flexibility. By being programmable, large computers got around this lack of flexibility, but their size has proved to be a handicap in that it is difficult to suitably isolate them from their industrial environment. Microprocessor-based controllers offer a flexible and packagable solution to the problem of industrial control. By their very nature they are small, which allows them to be easily isolated from the environment, and they are programmable, which makes them very adaptable to the varied tasks they may be required to perform.

This report presents tools which can be used in the development of software for these devices and, indeed, any application utilizing the MOS Technology 650x family of microprocessors and, optionally, the MOS Technology 6520 Peripheral Interface Adapter (PIA). The described software is a cross-assembler and simulator for the 650x

microprocessors, and a simulator for the 6520 PIA, which has been designed as an integral part of the 650x simulator.

ACKNOWLEDGEMENT

I would like to express my sincere thanks and appreciation to my thesis advisor, Professor Alfred Weaver, for his suggestions and encouragement during the course of this project, to the Department of Computer Science at the University of Illinois at Urbana-Champaign for funding the necessary computer work, and, finally, to my family, friends and colleagues for their continued encouragement and support.

TABLE OF CONTENTS

Chapter	Page
1. INTRODUCTION	1
2. THE 650x CROSS-ASSEMBLER	4
2.1. The Instruction Set	5
2.2. Assembler Directives	13
2.3. Cross-Assembler Use and Operation	19
3. THE 650x SIMULATOR	22
3.1. Static Commands	24
3.2. Run-time Commands	27
3.3. Exit Processing	42
3.4. Simulator Use and Operation	42
4. THE PERIPHERAL INTERFACE ADAPTER	46
4.1. PIA Commands	50
5. FUTURE WORK	54

REFERENCES 58

APPENDIX A: MCS650x Microprocessor Instruction Set 60

APPENDIX B: Instruction Addressing Modes 63

VITA 65

CHAPTER 1

INTRODUCTION

The advent of the microprocessor has offered a greatly expanded area of investigation for computer scientists and engineers alike. As a result of this technology, one of the many new applications which has become ripe for computerization is industrial process control. Throughout much of the 1960's industrial process control was dominated by relay logic. Various attempts to computerize different control functions using large (maxi or mini) computers were met with only partial success. One reason for the lack of immediate acceptance of this computerization is that controllers are often required to operate in very "non-computer-like" environments. In an industrial environment, such as a factory, a controller may be exposed to unfavorable temperature fluxuations, dust, oil mists, shock, electronic noise and vibrations, which have an adverse affect on the operation of electronic devices.

As an alternative to using large computers, a method which has been used with some degree of acceptance is that of solid-state discrete logic controllers. These solid-state controllers provide a packagable, fast alternative to both relay logic and larger computers. Due to its smaller size, a controller of this architecture can be fairly well isolated from the environment; however, these controllers inherit one of the major drawbacks of their relay logic predecessors, namely their lack of flexibility. Each solid-state or relay logic installation must be custom tailored to its application and any significant change in the industrial process may require costly, time-consuming hardware changes to be designed, tested and implemented.

As opposed to the rigid design of either relay or solid-state logic controllers, a microprocessor-based system provides a wide range of flexibility by being programmable and, due to its size, does not offer the drawbacks associated with larger computers. A typical microprocessor-based controller might be the size of a wall panel which can be mounted and suitably isolated from the environment.

As previously mentioned, one of the big advantages of microprocessor-based controllers is its flexibility. For example, if a process on some stage of an assembly line were changed, then, with a software driven system, the manufacturer would need to change only the program running in the controller rather than the actual controller hardware.

To effectively develop, modify and implement software for these programmable controllers, a software package has been developed which allows the user to develop, debug and test his/her software on a larger and more powerful machine. Once the user's program is operational, it can be loaded into, or the program ROM can be replaced in, the operational controller. Included in this package is a cross-assembler and simulator for the MOS Technology 650x family of microprocessors and a simulator of the MOS Technology 6520 Peripheral Interface Adapter (PIA). Among the many reasons why the MOS Technology products were chosen for this effort, three are especially important: (1) the 650x family provides the user with a fairly powerful addressing structure which in turn gives the user greater flexibility in programming style so as to best accomplish the control task; (2) the 650x family provides the user with this programming power at a relatively low cost, which is important from the business point of view; (3) the 650x family (and the 6502 in particular) is currently being used in industrial controllers such as the Struthers-Dunn "Director" series [1].

The format chosen for the remainder of this report is that of a series of user manuals. The decision to present the material about the software package in this manner was brought about by a desire to give users the greatest amount of useful understanding in a format that promotes easy future reference.

CHAPTER 2

THE 650x CROSS-ASSEMBLER

The 650x cross-assembler is the first piece of software in the MOS Technology 650x cross-assembler/simulator package. The cross-assembler is a series of FORTRAN routines implemented on the CDC Cyber 175 computer system at the University of Illinois at Urbana-Champaign. The following sections of this chapter discuss the instruction set, the assembler directives and the use of the cross-assembler.

2.1 The Instruction Set

The 650x family of microprocessors provides the user with a powerful set of machine instructions. This power is derived in part by the addressing structure available to the user, as well as the broad range of machine instructions. For a typical command, the command format is:

label opcode operands comments

With the exception of the opcode, all fields within this general form are optional depending upon the various requirements of the opcode; if any do appear, they must appear in the specified order and must be separated by at least one space.

2.1.1 Field Descriptions

The various rules pertaining to the fields for each of the instructions are as follows.

2.1.1.1 Labels

Labels are one to six alphanumeric characters, the first of which must be alphabetic. A second restriction placed on label names is that they may not be the same as any valid opcode or register name. The register names are A, X, Y, S and P. A list of the valid opcode mnemonics is found in Appendix A.

2.1.1.2 Opcodes

The opcode mnemonics are the three character alphabetic identifiers as shown in Appendix A. The machine instructions perform a wide variety of data handling, computational, testing, branching, status setting and interrupt control functions.

2.1.1.3 Operands

The operands are either an address, a constant value or an expression which can be evaluated at assembly time. Expressions are composed of symbolic names, constants and the operations add, subtract, multiply and divide (represented by +, -, * and / respectively). There is no operator precedence or parenthesizing. Strict left-to-right evaluation is implemented.

2.1.1.4 Comments

The comment field can have two forms. If the comment is part of a source line containing other valid fields, it must be the last field on the card and must be separated from the previous fields by at least one space. The second form occurs when the complete source line is to be treated like a comment, in which case the first non-blank character on the source line must be a semi-colon.

2.1.2 Addressing Modes

As mentioned previously, much of the power of 650x instruction set is derived from the powerful addressing modes available. There are eight modes of addressing: symbolic, immediate, relative, implied, page zero, indexed, indexed indirect and indirect indexed. Not all addressing modes may be used with every instruction. Appendix B shows which addressing modes are valid with each instruction.

2.1.2.1 Symbolic Addressing

Symbolic addressing is the addressing mode which is probably the most frequently used within a program. As the name implies, symbolic addressing occurs when the address used as an operand of an instruction is specified by a name. For example:

LDA JUNK

ADC SPUZ

STA CRUD

The above code sequence loads the accumulator with the contents of the memory location specified by JUNK, adds to it the contents of the memory location specified by SPUZ, and stores the results in the memory location specified by CRUD.

2.1.2.2 Immediate Addressing

The immediate mode of addressing specifies that the data needed for the operation specified is located in the memory byte following the opcode itself. The data must be a constant value and must be immediately preceded by a number sign (#). The constant values used as immediate data may be represented in any one of five forms - binary, octal, decimal, hexadecimal or Display Code (character). One difference between the version of the cross-assembler available here and any other is that immediate character data is treated as Display Code, NOT ASCII! For non-decimal values an escape character must precede the value. Table 1 shows the escape characters and their corresponding significance.

Escape Character	Defined Base
(none)	decimal
%	binary
@	octal
\$	hexadecimal
'	Display Code

Table 1

Constant Value Escape Characters

2.1.2.3 Relative Addressing

Relative addressing is used exclusively with the branch instructions. The relative address is, in effect, a positive or negative offset to the program counter from its current position. Since the program counter is automatically incremented during instruction execution, the program counter will point to the opcode following the branch instruction at the time the relative offset is applied. Therefore, the range of a branch is 127 bytes forward or 128 bytes backward from the program counter's current position. Any branch to an address outside these bounds is flagged as an error condition. For example:

BNE NOTZRO

causes the program to branch to the address specified by NOTZRO if the result of a previous compare does not produce a zero result. If the address NOTZRO is outside the specified bounds, then it will be flagged as an assembly time error.

2.1.2.4 Implied Addressing

Certain of the operations do not require any explicit declaration of their operands. The implied address of the required operands is inherent within the opcode itself. Examples of instructions which utilize implied addressing are the transfer instructions (TSX, TAY, etc.) which give both the source and destination of the data. Three other instructions are also treated in the same manner as implied

address instructions. These instructions (ASL, LSR, ROL) generate only a single byte opcode if the accumulator is specified as the sole operand.

2.1.2.5 Page Zero Addressing

One special form of address available on the 650x is page zero addressing. The 64K addressable memory space available to the microprocessor is referenced in 256 pages of 256 bytes each. The microprocessor has been designed such that, for instructions which reference a page zero address, a special opcode is used to signal the fact that only one byte of address is needed instead of the normal two. The advantages of this mode of addressing are that the user can generate faster and more compact programs by wisely allocating memory such that the most frequently referenced data areas are on page zero.

2.1.2.6 Indexed Addressing

The 650x family contains two index registers, X and Y. For instructions in which indexing by either one or both of these registers is valid, the first operand, immediately followed by a comma, and the index register name, X or Y, signals the indexed addressing mode to the cross-assembler. Indexed addressing takes the value stored in the specified register and adds it to the specified absolute address. The resulting address is the one used for the operation. For example:

```
LDA FIRST,Y
```

causes the accumulator to be loaded with the value pointed to by the sum of the address FIRST and the execution-time value in register Y.

2.1.2.7 Indexed Indirect Addressing

Indexed indirect addressing is much the same as indexed addressing, except the process is carried one step further. Once the indexed address is computed, it is used as an address to fetch the low and high bytes of the address of the location where the actual data is stored. It is extremely important to note that the computed indexed address used in the address fetch must point to a page zero location since during program execution the calculated high byte is ignored. Indexed indirect addressing is signaled by enclosing the address-register pair in parentheses as follows:

```
label  opcode  (address,index register)  comments
```

2.1.2.8 Indirect Indexed Addressing

As opposed to indexed indirect addressing, indirect indexed addressing performs the indirection prior to the indexing. Again, as was the case with the indexed indirect mode, the indirection can only be performed on a page zero address. An example of indirect indexed addressing is as follows:

```
LDA (LOC),X
```

In this example the accumulator is loaded with the contents of the

memory location specified by the low byte and high byte of the address stored at the page zero locations LOC and LOC+1 offset by the current value stored in the X register. It should be noted that there is no pure indirect addressing available except with the JMP instruction.

2.2 Assembler Directives

There are eight assembler directives available to the user. The various directives are used for the control of output generation and format, and for initializing variables and memory locations. Each directive (except for the equate) is signaled by a period followed by a keyword. The seven directives so represented are .BYTE, .WORD, .DBYTE, .PAGE, .SKIP, .OPT and .END. The equate is represented by an "equals" sign (=). A description of each of these directives follows.

2.2.1 .BYTE Directive

Form: label .BYTE value1,value2,...

Description: The .BYTE directive is used to assign a value, or values, to a specified byte or a group of contiguous bytes. The specified values may be any 8-bit constant, symbolic constant (see the section on the equate directive) or expression which has an 8-bit assembly time result. The label field is optional.

2.2.2 .WORD Directive

Form: label .WORD value1,value2,...

Description: The .WORD directive is specifically designed to load memory locations with addresses. Consequently, the 16-bit values specified are stored in the order low byte, high byte starting at the current address.

For example:

```
TABLE .WORD $FFFA,255
```

loads the locations TABLE, TABLE+1, TABLE+2 AND TABLE+3 with the values \$FA, \$FF, 255 and 0 respectively. Unlike the .BYTE directive, character strings are not valid values.

2.2.3 .DBYTE Directive

Form: label .DBYTE value1,value2,...

Description: As opposed to the .WORD directive, the .DBYTE directive stores the specified 16-bit quantities in the order in which they are specified. For example:

```
TABLE .DBYTE $FFFA,255
```

loads memory locations TABLE, TABLE+1, TABLE+2 and TABLE+3 with the values \$FF, \$FA, 0 and 255 respectively.

2.2.4 Equate Directive

Forms: (1) label *=*+value1

(2) *=value1

(3) symbolic constant=value1

Description: The `equ` directive, represented by the equal sign, has three functions. These functions are (1) to reserve a memory location or locations, (2) to assign a value to the program counter which is represented by an asterisk (*), and (3) to assign a value to a symbolic constant. In form 1, value1 number of bytes of memory are reserved. In the second form, the program counter is assigned the 16-bit address specified by value1. The third form assigns to the symbolic constant the 8-bit value specified. In each case, the value can be an expression which contains no forward references and can be resolved into a 16-bit (or 8-bit as the case may be) value at assembly time.

2.2.5 `.PAGE` Directive

Form: `.PAGE 'title'`

Description: The `.PAGE` directive is used in the formatting of the assembled listing on the listing file. The page command forces a page eject in the listing and prints the specified title. If no title is present, then the most recent title is repeated. As indicated on the form line above, the title is a character string enclosed in single quotes.

2.2.6 .SKIP Directive

Form: .SKIP value1

Description: The .SKIP command is used to generate blank lines in the listing file. Value1 is any valid constant or expression. If value1 is not specified, then .SKIP 1 is assumed.

2.2.7 .OPT Directive

Form: .OPT option1,option2,...option6

Description: The .OPT directive is used to set, or reset, up to six options. These options control the generation of the cross-reference listing, an error file, an instruction count, a listing file, an object file and the printing of Display Code strings used in .BYTE directives. Each of the options may be abbreviated by using the first three characters of the name. The options act as set/reset pairs and will be discussed as such.

2.2.7.1 XREF/NOXREF

Default Value: XREF

Description: The XREF/NOXREF options control the generation of a cross-reference listing. If the XREF option is set then a full cross-reference listing of the assembled program is generated.

2.2.7.2 ERRORS/NOERRORS

Default Values: ERRORS

Description: The ERRORS/NOERRORS options control the generation of a separate error file. If the ERRORS option is set then any error line in the source code is written to the separate error file along with the appropriate error message. The error message will also appear in the listing file. A complete description of the error messages and their meanings is given in [2].

2.2.7.3 COUNT/NOCOUNT

Default Value: NOCOUNT

Description: The COUNT/NOCOUNT options control the printing of the count of the number of times each instruction is used, the number of bytes of memory used, the number of symbols used and the number of source lines of input. If the COUNT option is selected all these counts will be written to the listing file. COUNT and NOCOUNT may also be abbreviated as CNT and NCNT respectively.

2.2.7.4 LIST/NOLIST

Default Value: LIST

Description: The LIST/NOLIST options control the generation of a listing file. If the LIST option is set then a listing file is generated which will contain the source code, the assembled code, error and warning messages, the symbol table, etc.

2.2.7.5 MEMORY/NOMEMORY

Default Value: MEMORY

Description: The MEMORY/NOMEMORY options are used to control the generation of the object file which is passed to the simulator. If the MEMORY option is set then the object file is generated.

2.2.7.6 GENERATE/NOGENERATE

Default Value: NOGENERATE

Description: The GENERATE/NOGENERATE options are use to control the printing of the Display Code strings in the .BYTE directives. If the GENERATE option is specified the full string is printed, otherwise just the first two characters are printed.

2.2.8 .END Directive

Form: .END

Description: The .END directive signals the cross-assembler that the physical end of the input source has been reached. This directive is optional, but its use is strongly recommended for completeness and documentation purposes.

2.3 Cross-Assembler Use and Operation

The implementation of the MOS Technology 650x Cross-Assembler on the CDC Cyber 175 computer system was designed for ease of use from the user's point of view. Two approaches will be used in the explanation of the use of the cross-assembler, (1) from the file structure point of view and (2) by example.

2.3.1 File Structure

The complete cross-assembler package is comprised of several files; however, only five are of interest to the user. These five files are a procedure file (ASM65), a source file (SOURCE), a listing file (LIST), an error file (ERROR) and an object file (OBJECT). The files contain exactly what their names imply; the source file has the input source code, the listing file contains the program listing as specified by the LIST option, the error file is the file generated by the ERRORS option and the object file contains the object code generated if the MEMORY option is set.

The cross assembler is invoked by inputting

```
GET,ASM65/UN=3AVN6V7
```

```
-ASM65
```

```
or
```

```
GET,ASM65/UN=3AVN6V7
```

```
-ASM65(SOURCE=s,LIST=1,ERROR=e,OBJECT=o)
```

where s, l, e and o are local files which contain or will contain

information which is normally found in the default file names. For example:

```
-ASM65(SOURCE=NEWPGM,ERROR=OUTPUT)
```

indicates that the source file is the local file NEWPGM, the listing file will be named LIST, the errors will be written to file OUTPUT (the user's terminal) and the object code will be written to file OBJECT. All files are rewound before and after use by the assembler.

2.3.2 A Sample Session

Throughout this short example is it assumed that the user has had some experience with the CDC Cyber 175 installation at the University of Illinois. In this example a program will be created using the BOSS editor, it will be assembled, the errors will be corrected, it will be reassembled and the listing file printed to the line printer.

Sample Session

```

/BOSS,TEST
BOSS 1.1.0
?? UNNUMBER
NO LINES
?? I
? ; THIS IS A TEST PROGRAM THAT...
? ; DOES NOTHING !
? ;
?      LDA #$FF
? LOOP  TAX
? LABEL1 DCX THIS IS AN ERROR
?      BNE LABEL1
?      JMP LOOP
?
?? E
FILE: TEST
$BOSS,TEST.
/GET,ASM65/UN=3AVN6V7
/-ASM65(SOURCE=TEST,ERROR=OUTPUT)
  7  LABEL1 DCX THIS IS AN ERROR
***** ERROR ** ILLEGAL OR MISSING OP CODE - NEAR COLUMN 10

```

END OF MOS/TECHNOLOGY 650X ASSEMBLY VERSION 7.0

```

NUMBER OF ERRORS=      1,  NUMBER OF WARNINGS=      0
*RUN COMPLETE.
/BOSS,TEST
?? F/DCX/
LABEL1 DCX THIS IS AN ERROR
?? M
    LABEL1 DCX THIS IS AN ERROR
M??      E      ^NOT #
LABEL1 DEX THIS IS NOT AN ERROR
?? E
FILE: TEST
$BOSS,TEST.
/RWF
RWF FINISHED.
/-ASM65(SOURCE=TEST)
*RUN COMPLETE.
/PRINT,LIST/CC/EJ

```

CHAPTER 3

THE 650x SIMULATOR

The 650x simulator is the second piece of software included in the cross-assembler/simulator package. The simulator represents, to a very good approximation, the execution of the 650x family of MOS Technology microprocessors. Overlaid over the basic simulator structure is a relatively powerful monitor which provides user control over both program execution and the environment in which the program resides. The following sections in this chapter present a fairly extensive description of each of the commands available, as well as a description of the use of the simulator on the CDC Cyber 175 computer system at the University of Illinois at Urbana-Champaign.

The notational conventions for the various constant values will be the same as described in Chapter 2 in the section on immediate addressing. In addition, the following conventions are also followed. Any item prefixed by 'loc' is a 16-bit location, any item prefixed by 'reg' is a register name, any item prefixed by 'value' is an 8-bit value

and any item prefixed by 'page' is a valid hexadecimal page number (\$0 - \$FF). Unless otherwise noted, any numeric input may be in base 2, 8, 10 or 16, and may optionally be a symbolic name defined in the cross-assembled program.

3.1 Static Commands

The static commands are a series of eight monitor level instructions which allow the user to specify the environment in which the program is to execute. A description of these commands follows.

3.1.1 Set Memory

Command Form: SM loca,value1,value2,...,value16

Description: The set memory command allows the user to load values into up to sixteen consecutive memory locations starting at location loca.

Example: SM \$FFFA,\$10,20,@30,%10110,\$1A,\$FF

The above command loads memory locations hexadecimal FFFA to FFFF with the values of hexadecimal 10, decimal 20, octal 30, binary 10110, hexadecimal 1A and hexadecimal FF respectively.

3.1.2 Set Register

Command Form: SR reg1,value1,reg2,value2,...,reg5,value5

Description: The set register command allows the user to statically set register values. Up to five registers (corresponding to the A, X, Y, S and P registers) may be specified.

Example: SR A,%101,S,\$50

The above command sets the A register with binary 101 and the S register with hexadecimal 50.

3.1.3 Memory Definition

Command Form: MD loca,locb,mtype

Description: When the system is initialized, the entire 64K of memory space is defined to be RAM. The memory definition command allows this initial specification to be changed according to the user's needs and wishes. More specifically, the use of this command allows the block of memory bounded by loca and locb to be define as mtype where mtype is either RAM, ROM or I/O. Memory definition commands are effective only if the initial memory definition has been cleared.

Example: MD 0,\$DFFF,RAM

MD \$E000,\$EFFF,I/O

MD \$F000,\$FFFF,ROM

The above three memory definitions used in conjunction with each other define the first hexadecimal E000 locations to be RAM, the next hexadecimal 1000 locations to be I/O addresses and the last hexadecimal 1000 locations to be ROM. A maximum of 25 memory definition commands may be in effect at a time.

3.1.4 Clear Memory Definition

Command Form: CMD

Description: The clear memory definition command undoes the effect of all previous memory definiton commands. This command does not reestablish the initial condition of all 64K of memory space being defined as RAM.

3.1.5 Memory Wiring Definition

Command Form: MWD page1,page2

Description: The memory wiring definition command allows the user to specify the memory wiring of the microprocessor. Illustrating by example, if the user issued the command 'MWD \$0,\$1', then all references to page 1 of the microprocessor's memory space would actually reference page 0. Due to this effective mapping of page 1's memory space, any DUMP of an address on page 1 will actually display the contents of the corresponding memory location on page 0 while this particular Memory Wiring Definition command is in effect. As should be noted from the example, the page numbers specified must be hexadecimal constants.

3.1.6 Clear Memory Wiring Definition

Command Form: CMWD

Description: The clear memory wiring definition command undoes the effect of all previous memory wiring definition commands by reestablishing the default condition of each page corresponding to itself.

3.1.7 Static TRACE

Command Form: TRACE

Description: The static trace command is specifically designed to allow the user to examine the current values stored within the registers. For

consistency, the same format of trace output is maintained here as in the dynamic trace; hence, other information such as the current instruction count, the program counter, etc. are also displayed.

3.1.8 Static DUMP

Command Forms: (1) DUMP loca

(2) DUMP loca,locb

Description: The static dump commands provides the user with a memory dump of either a single location, loca, or the block of locations from loca through locb. All dumps are displayed in a hexadecimal format.

3.2 Run-time Commands

The many run-time commands available in the 650x simulator fall into four categories - control, setup, special control and run-block commands. The following sections will describe each of these categories at length.

3.2.1 Run-time Control

The run-time control commands provide the user with the capability of selecting, or deselecting, various break-point conditions during program execution. The commands in this category are represented by single, nonparameterized keywords. With the exception of the Clear

Controls command, all commands exist as set/reset pairs and will be discussed as such.

3.2.1.1 Break/No Break on any Error

Command Forms: BER

NBER

Default Value: BER

Description: The BER/NBER commands allow the user to determine what action is taken when one of the three following error conditions are detected.

(1) Invalid op code

(2) Memory access - address not in defined region

(3) Memory access - attempt to write into ROM

If BER is set, then when any of the above conditions occur, a break-point is forced and the simulator returns to the monitor level and awaits further action by the user. If NBER is in effect, then no break-point is forced, and execution continues in spite of the error.

3.2.1.2 Break/No Break on Interrupt Return

Command Forms: BIR

NBIR

Default Value: NBIR

Description: If BIR is set, a break-point is forced whenever an RTI instruction is executed within the user's program. A NBIR command negates the effect of a previous BIR commands and allows execution to be continuous on interrupt returns.

3.2.1.3 Break/No Break on Masked Interrupt

Command Forms: BMI

NBMI

Default Value: BMI

Description: When the BMI command is set, a break-point is forced if a simulated IRQ occurs and interrupts are masked. The NBMI allows execution to be continuous in spite of the occurrence of this condition.

3.2.1.4 Break/No Break on Subroutine Return

Command Forms: BSR

NBSR

Default Value: NBSR

Description: When BSR is set, a break-point is forced if an RTS instruction is executed within the user's program. The NBSR command allows for continuous execution on subroutine returns.

3.2.1.5 Break/No Break on any Warning

Command Forms: BW

NBW

Default Value: NBW

Description: If the BW command is set, a break-point will be forced when any of the following warning conditions occur.

- (1) Page zero wrap - the effective address calculated for a zero page instruction has gone from hexadecimal FF to 0.
- (2) Hexidecimal page FF wrap - the program counter has wrapped from hexadecimal address FFFF to 0.
- (3) Page crossed (absolute addressing) - the data value addressed resides on a different page, requiring an additional cycle to calculate the effective address.
- (4) Page crossed (branch instruction) - The address to which the branch is to occur is on a different page and requires an additional machine cycle to calculate.
- (5) Decimal mode (non-BCD value) - The 4-bit input digit required for a decimal arithmetic operation is a non-BCD character.

3.2.1.6 Clear Controls

Command Form: CC

Description: The CC command is used to globally reestablish all the default settings of the above control commands.

3.2.2 Run-time Setups

The run-time setup commands afford the user the ability to set, trace or dump the registers and/or memory locations during program execution. Also included in this category are the interrupt commands which allow the user to force a break-point at various points within his/her program (by simulating an IRQ or NMI) and perform the set, trace and dump functions statically.

3.2.2.1 BEFORE

The BEFORE command has six alternate forms which fall into two general categories; listed-directed and incremental. The function of the BEFORE command is to allow a memory location or register to be dynamically set to a specific or relative value before the instruction at the specified location is executed. A maximum of twenty BEFORE commands may be in effect at a given time. A description of the various forms of this command is given below.

3.2.2.1.1 List-directed BEFORE

Command Forms: (1) BEFORE loca, locb=value1,value2,...,value10

(2) BEFORE loca, reg1=value1,value2,...,value10

Description: Prior to the first execution of the instruction at loca, memory location locb (form 1) or register reg1 (form 2) will be assigned the value value1. Thereafter, before each subsequent execution of the

instruction at location loca, the memory location or register will be assigned the next value in the list. Up to ten values may be specified.

Examples:

(1) BEFORE LABEL1,\$FCAA=%1010,%0101,%1010

Before the instruction at LABEL1 (a symbolic value defined in the assembled program) is executed for the first time, hexadecimal location FCAA will be assigned the binary value 1010. On the second time the instruction at LABEL1 is executed location FCAA will have the binary value of 0101 and on the third and subsequent times FCAA will have the value 1010.

(2) BEFORE \$2FF,X=1,2,3,5,8

Before the instruction at hexadecimal location 2FF is executed for the first time, register X will be assigned the value 1. On subsequent executions of the instruction at location 2FF register X will be assigned the values 2, 3, 5 and 8.

3.2.2.1.2 Incremental BEFORE

Command Forms: (1) BEFORE loca,locb=locc.+value1

(2) BEFORE loca,reg1=locc.+value1

(3) BEFORE loca,locb=reg1.+value1

(4) BEFORE loca,reg1=reg2.+value1

Description: In these four forms of the BEFORE command, prior to the execution of the instruction loca, memory location locb (forms 1 and 3) or register reg1 (forms 2 and 4) is assigned the value stored in memory

location locc (forms 1 and 2) or register reg2 (forms 3 and 4) plus the constant value value1.

Examples:

(1) BEFORE LABEL,HIGH=LOW.+10

Before the instruction at LABEL (a symbol defined in the assembled program) is executed, the memory location specified by HIGH, another program defined symbol, is assigned the value stored in LOW incremented by 10.

(2) BEFORE LABEL,X=LOW.+10

Same action as example 1, except that register X is assigned the value of LOW incremented by 10.

(3) BEFORE \$AOFA,\$FF00=X.+@10

Before the instruction at hexadecimal location AOFA is executed, hexadecimal location FF00 is assigned the value stored in register X incremented by an octal 10.

(4) BEFORE \$AOFA,X=X.+@10

Same as example 3, except X is assigned its old value incremented by an octal 10.

3.2.2.2 Dynamic DUMP

The user has three options as to when a memory dump can be generated during program execution. These occur when the instruction address has a particular value, when the calculated effective address has a particular value, and when execution terminates at the end of a run-block. Up to twenty dynamic dumps may be specified for a run-block

execution (run-blocks are discussed in a future section). A detailed description of the DUMP commands follows.

3.2.2.2.1 Instruction Address DUMP

Command Form: DUMP loca,locb,.IA=locc

Description: Each time the instruction address attains the value of locc, the contents of memory from loca through locb is dumped to the user's terminal.

3.2.2.2.2 Effective Address DUMP

Command Form: DUMP loca,locb,.EA=locc

Description: Every time the calculated effective address becomes equal to the value locc, the region loca through locb is dumped.

3.2.2.2.3 End of Run-Block DUMP

Command Form: DUMP loca,locb,.END

Description: Upon the successful completion of the current run-block a memory dump of the locations loca through locb is generated.

3.2.2.3 Dynamic TRACE

The dynamic TRACE command allows the user to dynamically watch program execution. The traces provide the user with current information

about the instruction just executed, the state of the condition bits in the status word, and the current effective address, program counter, instruction count and execution time. There are four types of dynamic traces available to the user which are discussed below. Up to twenty dynamic traces may be specified for a run-block.

3.2.2.3.1 Single Address TRACE

Command Form: TRACE loca

Description: After the instruction at location loca is executed a trace line is generated.

3.2.2.3.2 Block Address TRACE

Command Form: TRACE loca,locb

Description: When an instruction located within the bounds of loca and locb is executed, a trace line is generated.

3.2.2.3.3 Effective Address TRACE

Command Form: TRACE .EA=loca

Description: When the calculated effective address becomes equal to loca a trace line is generated after the instruction is executed.

3.2.2.3.4 End of Run-Block TRACE

Command Form: TRACE loca,.END

Description: The trace will commence each time the instruction address becomes greater than or equal to the location loca. The trace will continue as long as that lower bound is maintained through the end of the execution of the run-block.

3.2.2.4 Interrupt-at

Command Form: IAT loca

Description: After the instruction at loca is executed an interrupt request (IRQ) is simulated causing the program to jump to the address specified in hexadecimal locations FFFE and FFFF. The interrupt request can be masked by disabling interrupts within the user's program. If interrupts are masked and an interrupt occurs, an optional break-point may be forced (cf. Section 3.2.1.3).

3.2.2.5 Non-Maskable Interrupt-at

Command Form: NMIAT loca

Description: After the instruction at location loca is executed, a non-maskable interrupt (NMI) is simulated. The occurrence of a non-maskable interrupt will cause the program to jump to the address stored in hexadecimal locations FFFA and FFFB.

3.2.2.6 Clear Setups

Command Form: CSU

Description: The clear setups command clears the dynamic setups caused by the BEFORE, DUMP, TRACE, interrupt-at and non-maskable interrupt-at commands.

3.2.3 Special Run-time Controls

There are three commands which fall into the category of special run-time controls. These commands pertain to the maximum amount of time or maximum number of instructions which can be executed before a break-point is forced. These three commands are TIME, MAXTIME and MAXSTEPS. The MAXSTEPS and MAXTIME commands are implemented in such a way that only one of the two may be in effect at any given time. This mutual exclusion is accomplished by allowing only the most recently specified command to maintain control.

3.2.3.1 Time Control

As previously mentioned, the user may specify the maximum amount of simulated time which his/her program may execute. The TIME and MAXTIME commands used in conjunction with each other enable this facility. A description of these commands follows.

3.2.3.1.1 TIME

Command Forms: (1) TIME=t

(2) TIME=tUSECS

Default Value: 0.0

Description: The time command determines the timebase which is to correspond to one microprocessor cycle. The parameter t is a five character decimal number optionally including a decimal point. The appending of the string 'USECS' is optional and redundant since all specified times are in micro-seconds.

3.2.3.1.2 MAXTIME

Command Form: MAXTIME=t

Default Value: 0.0

Description: If the MAXTIME command is in effect, then a break-point will be forced if, after the completion of the execution of the current instruction, the total execution time is greater than the specified value. As is the case with the TIME instruction, t is a five digit decimal number, optionally including a decimal point, which specifies the maximum number of micro-seconds execution is to continue.

3.2.3.2 MAXSTEPS

Command Form: MAXSTEPS=m

Default Value: 1000

Description: If the MAXSTEPS command is in effect, a break-point will be forced if the instruction count becomes greater than the specified value *m*. The parameter *m* is a five digit decimal integer.

3.2.4 Run-Block Commands

The run-block commands are the ones which actually cause the execution of the program or any block of code. These commands fall into two groups specified by the keywords DO and START. These commands are discussed below.

3.2.4.1 DO Commands

There are six general forms of the DO command which give the user a good range of control over the execution of a block of code.

3.2.4.1.1 Simple Block

Command Form: DO loca,locb

Description: This form of the DO command specifies that the instructions starting at location loca and continuing through the instruction at location locb are to be executed.

.

3.2.4.1.2 Maximum Value

Command Forms: (1) DO loca,locb=value1

(2) DO loca,reg1=value1

Description: For these forms of the DO command execution is started with the instruction at location loca and continues until memory location locb (form 1) or register reg1 (form 2) attains the value of value1.

3.2.4.1.3 Simple Block with Value Limiting

Command Forms: (1) DO loca,locb,locc=value1

(2) DO loca,locb,reg1=value1

Description: Execution starts with the instruction at location loca and continues through the instruction at location locb, unless location locc (form 1) or register reg1 (form 2) attains the value specified by value1.

3.2.4.1.4 Simple Block with Address Limiting

Command Forms: (1) DO loca,locb,.IA=locc

(2) DO loca,locb,.EA=locc

Description: Execution will commence with the instruction at location loca and continue through the instruction at location locb, unless the instruction address (form 1) or the effective address (form 2) becomes equal to the specified location locc.

3.2.4.1.5 Multiple Iteration

Command Form: DO loca,locb value1.TIMES

Description: This form of the DO command causes the block of code starting at location loca and continuing through location locb to be repetitively executed value1 number of times.

3.2.4.1.6 Resumption after a Break

Command Form: DO

Description: After a break-point has occurred (except if it has been caused by the completion of a run-block) execution can continue with the next instruction in the user's program by simply inputting 'DO' when the command request is made.

3.2.4.2 START Commands

Command Forms: The START command has all the same command forms as the DO commands discussed in the previous sections, except there is no form to resume execution after a break.

Description: The START commands commences execution by simulating activation of the microprocessor's RESET line. A START command causes the program to jump to the address specified in hexadecimal locations FFFC and FFFD and begin execution of the code at that address. These commands otherwise process the run-block in the same way as the corresponding DO commands.

3.3 Exit Processing

Command Form: EXIT

Description: The EXIT command causes the user to exit from the simulator monitor and, ultimately, exit from the simulator itself. Control is returned to the Cyber operating system.

3.4 Simulator Use and Operation

The use of the simulator on the CDC Cyber 175 computer system at the University of Illinois is actually quite simple. This ease of use is made possible in part through the use of a procedure file SIM65. The next two subsections will take a look at the simulator from two aspects, the files involved and a sample execution session. As was the case with the cross-assembler, it is assumed the user has a working knowledge of the local Cyber installation.

3.4.1 The Files

There are three files requiring user familiarity. These files are the procedure file (SIM65), the object file from the cross-assembler (OBJECT), and the virtual memory file (VMEM). The procedure file and the object file are both sequential files. The virtual memory file, VMEM, is a mass storage file. As a result of its file structure, the user must be certain that if simulator execution is abnormally terminated (either by user or system action), the virtual memory file is

RETURNed before attempting to reinvoke the simulator. Failure to return the file will cause a fatal error since an abnormal termination does not close the file.

The simulator is invoked by inputting:

```
GET,SIM65/UN=3AVN6V7
```

```
-SIM65
```

or

```
GET,SIM65/UN=3AVN6V7
```

```
-SIM65(OBJECT=o,VMEM=v)
```

where o and v are local files of the proper type. If a local virtual memory file does not exist, then one will be created by the system.

3.4.2 A Sample Session

In this sample session a program will be created using the BOSS editor, then assembled and its object code loaded into the simulator. The source code for the assembler is on file SAMPLE. The files for the simulator have the default names of OBJECT and VMEM.

Sample Session

```

/BOSS,SAMPLE
BOSS 1.1.0
?? UNNUMBER
NO LINES
?? I
? ; THIS IS A SAMPLE PROGRAM TO CALCULATE THE SQUARE OF THE NUMBER
? ; AT LOCATION INPUT BY REPEATED ADDITION AND PLACE THE RESULT
? ; AT LOCATION OUTPUT.
? ;
? INPUT .BYTE 3
? OUTPUT .BYTE 0
? *=$200
? START LDX INPUT
? BEQ DONE
? LDA #0
? LOOP ADC INPUT
? DEX
? CPX #0
? BNE LOOP
? DONE STA OUTPUT
? HALT JMP HALT
?
?? E
FILE: SAMPLE
$BOSS,SAMPLE.
/GET,ASM65/UN=3AVN6V7
/-ASM65(SOURCE=SAMPLE,ERROR=OUTPUT)

```

END OF MOS/TECHNOLOGY 650X ASSEMBLY VERSION 7.0

```

NUMBER OF ERRORS=      0,  NUMBER OF WARNINGS=      0
*RUN COMPLETE.
/GET,SIM65/UN=3AVN6V7
-SIM65

```

MOS TECHNOLOGY 650X SIMULATOR - VERSION 10.062877-15

OLD OR NEW VIRTUAL MEMORY FILE?

? NEW

VIRTUAL MEMORY INITIALIZED.

LOADING OBJECT FILE.

COMMAND-

? DO START,HALT

IA	LABEL	OPCODE	A	S	X	Y	P	STATUS	PC	EA	EO	ICNT	TCNT	TIME
T020A	HALT	JMP 4C	0C	00	00	00	13	B	ZC 0211	0211	00	18	44	0.00

HILEV BREAKPOINT-NORMAL DO SEQUENCE END

COMMAND-

? EXIT

EXIT.

CHAPTER 4

THE PERIPHERAL INTERFACE ADAPTER

The MOS Technology 6520 Peripheral Interface Adapter (PIA) provides the system designer with a uniform and relatively simple interface between the 650x family of microprocessors and a host of peripheral devices. There are two main design features of the PIA which are primarily responsible for this uniformity and simplicity. These features are (1) the use of the PIA obviates the need for additional external logic when interfacing a 650x microprocessor with most peripheral devices, and (2) the PIA contains four user programmable registers to control the flow of data and control signals.

On a macroscopic level, the PIA consists of three 8-bit bi-directional data ports. One of these ports is the 8-bit data bus to the microprocessor and one of the other two goes to each peripheral device. Besides three data ports, there are also three groups of control lines which again have one group going to the microprocessor and

one to each peripheral device. In addition to the control and data lines, there are six user addressable Registers - the Data Direction Registers (DDRA and DDRB), the Control Registers (CRA and CRB) and the Peripheral Output Registers (ORA and ORB). The suffix A or B indicates which peripheral device the register serves or controls.

As previously mentioned, four of the PIA registers are user programmable. These registers are the two Data Direction Registers and the two Control Registers. As the name may indicate, the Data Direction Registers control the direction of the data on each of the lines of the peripheral data ports. A "0" in the corresponding bit position of the Data Direction Register causes the data line to act as an input line, and a "1" causes it to act as an output.

The output registers ORA and ORB utilize the information in the Data Direction Registers. The output from the PIA data ports represents the logical product of the output register and its corresponding data direction register.

The Control Registers allow the microprocessor to control the operation and effect of the control lines CA1, CA2, CB1 and CB2 as well the addressing of the Data Direction and Output registers. Nothing short of an extensive discussion, such as the one given in [3], could fully explain the use of the control registers and the control lines; however, a summary of the functions of each of the bits in CRA is given in Table 2. The function of the bits in CRB is analogous.

- Bit 0 - 1 to disable CA1
- 0 to enable CA1
- Bit 1 - 1 if an active transistion of CA1 is to be positive
- 0 if an active transistion of CA1 is to be negative
- Bit 2 - 1 to address ORA
- 0 to address DDRA
- Bit 3 - 1 to disable CA2
- 0 to enable CA2
- Bit 4 - 1 if an active transistion of CA2 is to be positive
- 0 is an active transistion of CA2 is to be negative
- Bit 5 - 1 if CA2 is to be used as a peripheral control line
- 0 if CA2 is to be used as a peripheral interrupt line
- Bit 6 - 1 if interrupt request from CA2
- 0 otherwise
- Bit 7 - 1 if interrupt request from CA1
- 0 otherwise

Table 2

CRA Bit Functions

To address these six accessible registers from within a user program, the user must allocate a block of four contiguous bytes in which the PIA is to be located and, once the program has been assembled and loaded into the simulator, the corresponding area must be defined as a PIA. A modification and extension has been made to the current memory definition command for this purpose. Within the four bytes of memory, any reference to the first byte will reference Peripheral A (actually ORA on a store) or DDRA depending on the status of bit 2 in CRA. The second byte within the area corresponds to CRA, the third byte peripheral B (ORB on stores) or DDRB depending on CRB bit 2 and the fourth byte corresponds to CRB.

The simulation of the PIA was designed in the same style as the 650x simulator so as to provide the user with a single, powerful, cohesive industrial process control package. In the spirit of industrial process control, the outputs from the the PIA are a binary image of the eight bits in each of the two output buffer registers. Also in binary are the PIA traces which will be discussed in a later section of this chapter. To improve the human interface of the simulation, the user inputs to the PIA may be in base 2, 8, 10, 16 or character (Display Code). The inputs must be explicit (ie. not symbolic) constants and follow the convention of escape characters specified in Table 1 of Chapter 2. Input data is free format and the proper number of bits will be collected until the end of the input line, a space or an invalid character for the given base is detected.

4.1 PIA Commands

The PIA commands provide the user with the means of defining a PIA to be located within the microprocessor's memory space, examining the PIA's internal registers and controlling the states of the various control lines. The following sections give a description of the commands.

4.1.1 Memory Definition

Command Form: MD loca,PIA

Description: The memory definition command allows the user to define an area of four contiguous bytes of memory as PIA addresses. This command is an extension and modification of the memory definition command already available in the 650x simulator (cf. Section 3.1.3). Currently only one PIA may be defined at any given time. The memory definition command is overridden using the Clear Memory Definition (CMD) command described in Section 3.1.4.

4.1.2 Static Trace

Command Form: PIAT

Description: The static trace provides the user with an immediate trace of the six internal PIA registers within a static environment. The contents of these six registers, namely the control registers CRA and CRB, the data direction registers DDRA and DDRB, and the output registers ORA and ORB, are displayed in an 8-bit binary image format.

4.1.3 Dynamic Trace

Command Form: PIAT *

Description: The dynamic trace allows the user to inspect the contents of the six internal PIA registers as the user program executes. If the dynamic trace is in effect, then a trace of the six internal PIA registers will be generated whenever processing occurs inside the PIA.

4.1.4 Interrupt Control and Processing

The PIA is the interface between two or more asynchronous interrupt-driven devices. To accomplish its function, the PIA has six interrupt lines; two control lines to/from peripheral device A, two control lines to/from peripheral device B and two interrupt request lines from the PIA to the microprocessor. As previously mentioned, a complete description of the PIA and its operation (including the interrupt structure) is given in [3] and will not be repeated in this manual. To provide the user with maximum control of the interrupt process, a series of new commands has been added to the 650x simulator which allow the user to control both the type of interrupt generated by each peripheral device (i.e., IRQ=maskable or NMI=nonmaskable) and the state of the four control lines at any given location during program execution.

4.1.4.1 Interrupt Type Selection

Command Forms: (1) PIAI periph=IRQ

(2) PIAI periph=NMI

Default Value: Peripheral A = Peripheral B = IRQ

Description: The PIA interrupt (PIAI) commands allow the user to simulate the wiring of the interrupt request lines IRQA and/or IRQB to the NMI or IRQ pins on the microprocessor. By default, both interrupt request lines will simulate an IRQ when the proper states of the control lines cause either line to go low (active). In the above command forms, periph is either the single character A or B, representing peripheral device A or B respectively.

4.1.4.2 Control Line Processing

Command Forms: (1) CA1 loca,value1

(2) CA2 loca,value1

(3) CB1 loca,value1

(4) CB2 loca,value1

Description: Following the interrupt scheme developed and implemented in the 650x simulator, the PIA control line handling was designed and implemented in the same manner. The four commands shown above allow the user to specify the value that the designated control line will have at the time the program counter reaches the address specified by loca. The value value1 is, for the purposes of the simulation, taken to be a logic 0 or 1; hence, any non-zero value specified is assumed to be a logic one.

Whenever any of the above commands is issued, the 650x simulator schedules the type of interrupt selected by the PIAI command at the specified location. If, when the program counter achieves the address specified by loca, the action of the control line causes the proper interrupt request line to go low, then the proper type of interrupt is processed in the manner described in Chapter 3.

CHAPTER 5

FUTURE WORK

The entire industrial process control package described in this report has been implemented and is fully operational on the CDC Cyber 175 computer system at the University of Illinois at Urbana-Champaign. The package has been exercised extensively both in the course of testing and through production use; all known errors have been corrected.

Though the software package in itself is complete, there are several things which could be done to enhance it. These enhancements, though not extremely difficult to implement, were not included in the current versions of the programs due to time limitations. Between the cross-assembler and the simulator there are eight suggested modifications. The first six changes apply to both the cross-assembler and the simulator, and the last two apply exclusively to the simulator.

- (1) The standard character representation on the CDC machine is 6-bit Display Code. It would be desirable to alter the

necessary routines in both programs to convert all literal constant or input data (flagged by a single quote (')) from Display Code to ASCII. This change is especially desirable when a literal is used as immediate data for a machine instruction in a user program.

- (2) Throughout the FORTRAN code of both the simulator and cross-assembler programs an effort should be made to improve the use of memory. This suggestion arises from the fact that the cross-assembler, for instance, requires over 120K 60-bit words to load.
- (3) Closely related with the memory management issue is that of optimized execution times. There are several places where, judging from hindsight rather than foresight, code sequences could be improved to provide for faster program execution. Due to the speed of the central processor on the Cyber the response time is generally acceptable, but faster (ie., more efficient) code would make a better product.
- (4) Both the cross-assembler and the simulator are comprised of many subroutines and common blocks. Before changes of any consequence are made to either of these programs serious thought should be given to the idea of creating program libraries to facilitate separate compilation of the subroutines and enforce consistency among the common blocks. Control Data provides two possible alternatives for solving this problem on

the Cyber, one being UPDATE and the other MODIFY. UPDATE [4] and MODIFY [5] are utility programs available for creating and updating (or modifying) program source libraries.

- (5) Another possible feature which has been suggested on several occasions is interactive file handling. This modification would require writing (or usurping) COMPASS routines which are able to create and manipulate permanent files from within the FORTRAN programs.
- (6) A feature which will have more usefulness in the simulator, but is applicable to both programs, is the ability to detect interrupts from the user's terminal. If, for example, the program executing gets into a long loop which the user wishes to terminate, then by simply typing 's' (or another suitable character) the user would be able to halt what is currently being executing and cause the program to jump to some specified point within the program. In the simulator, for example, the logical place to return to would be the command interpreter (CMNDI).
- (7) This suggested modification, and the following one, pertain exclusively to the simulator. Currently the interrupt structure is based upon an IRQ or an NMI being scheduled to occur when the program counter attains a certain value. To greatly expand the interrupt capabilities and provide a more "real-world" environment the capability should be added to

allow interrupts to occur at times internally selected by some random means or in response to a keypress at the user's terminal. This should be done in addition to, not in place of, the current interrupt structure.

- (8) This final suggested modification is quite simple. The current design of the simulator is capable of handling only one PIA. This final modification would be to upgrade the simulator to permit multiple PIAs.

These suggested changes are not ordered by priority, nor do they represent an exhaustive list. The software package was designed to give the user a great deal of freedom of use and to allow him/her to use this package as a stand-alone facility or in conjunction with another programming design effort.

REFERENCES

- [1] Weaver, Alfred C., "Architecture and Operating System for the Struthers-Dunn Director 1001 Controller", April 1977, unpublished.
- [2] "MCS6500 Microcomputer Family Cross-Assembler Manual - Preliminary", MOS Technology, Inc., Norristown, PA 19401, 1975.
- [3] "MCS6500 Microcomputer Hardware Manual", MOS Technology, Inc., Norristown, PA 19401, 1976.
- [4] "NOS Version 1 UPDATE Reference Manual", Control Data Corporation, 1976.
- [5] "NOS Version 1 MODIFY Reference Manual", Control Data Corporation, 1976.
- [6] "MCS6500 Microcomputer Family Cross-assembler Manual - Preliminary", page 4.
- [7] "MCS6500 Microcomputer Family Cross-assembler Manual - Preliminary", page 12.
- [8] Weaver, Alfred C., "A Graphically-Programmed, Microprocessor-Based Industrial Controller", Ph.D. Thesis, Department of Computer Science Report No. UIUCDCS-R-77-865, University of Illinois, Urbana, Illinois 61801, May 1977.

- [9] Weaver, Alfred C., class notes for "Microprocessor Architecture and Software Systems", University of Illinois, Urbana, Illinois 61801, Spring 1977, unpublished.

- [10] "MCS6500 Microcomputer Simulator Manual - Preliminary", MOS Technology, Inc., Norristown, PA 19401, 1975.

- [11] "MCS6500 Microcomputer Software Manual", MOS Technology, Inc., Norristown, PA 19401, 1976.

APPENDIX A

MCS650x Microprocessor Instruction Set [6]

ADC	Add with Carry to Accumulator
AND	Logical AND to Accumulator
ASL	Shift Left One Bit (Memory or Accumulator)
BCC	Branch on Carry Clear
BCS	Branch on Carry Set
BEQ	Branch on Zero Result
BIT	Test Bits in Memory with Accumulator
BMI	Branch on Minus Result
BNE	Branch on Result not Zero
BPL	Branch on Result Plus
BRK	Force an Interrupt or Break
BVC	Branch on Overflow Clear
BVS	Branch on Overflow Set
CLC	Clear Carry Flag
CLD	Clear Decimal Flag
CLI	Clear Interrupt Disable Bit
CLV	Clear Overflow Flag
CMP	Compare Memory with Accumulator
CPX	Compare Memory with Index X
CPY	Compare Memory with Index Y
DEC	Decrement Memory by One

DEX	Decrement Index X by One
DEY	Decrement Index Y by One
EOR	Exclusive-or Memory with Accumlator
INC	Increment Memory by One
INX	Increment X by One
INY	Increment Y by One
JMP	Jump to New Location
JSR	Subroutine Call
LDA	Transfer Memory to Accumlator
LDA	Transfer Memory to Index X
LDY	Transfer Memory to Index Y
LSR	Shift One Bit Right (Memory or Accumlator)
NOP	No Operation
ORA	Logical OR Memory with Accumlator
PHA	Push Accumlator on Stack
PHP	Push Processor Status on Stack
PLA	Pull Accumlator from Stack
PLP	Pull Processor Status from Stack
ROL	Rotate One Bit Left (Memory of Accumlator)
RTI	Return from Interrupt
RTS	Return from Subroutine
SBC	Subtract Memory and Carry from Accumlator
SEC	Set Carry Flag
SED	Set Decimal Mode

SEI	Set Interrupt Disable Status
STA	Store Accumulator in Memory
STX	Store Index X in Memory
STY	Store Index Y in Memory
TAX	Transfer Accumulator to Index X
TAY	Transfer Accumulator to Index Y
TSX	Transfer Stack Register to Index X
TXA	Transfer Index X to Accumulator
TXS	Transfer Index X to Stack Register
TYA	Transfer Index Y to Accumulator

APPENDIX B

Instruction Addressing Modes [7]

IM = Immediate

PZ = Page Zero

AB = Absolute

ZX = Page Zero Indexed by X

AX = Absolute Indexed by X

AY = Absolute Indexed by Y

II = Indexed Indirect

Opcode -----	IM --	PZ --	AB --	ZX --	AX --	AY --	II --
ACD	X	X	X	X	X	X	X
AND	X	X	X	X	X	X	X
ASL (1)		X	X	X	X		
BIT		X	X				
CMP	X	X	X	X	X	X	X
CPY	X	X	X				
CPX (2)	X	X	X				
DEC		X	X	X	X		
EOR	X	X	X	X	X	X	X
INC		X	X	X	X		
JMP (3)			X				X
JSR			X				

LDA	X	X	X	X	X	X	X
LDX (2)	X	X	X	X	X		
LDY	X	X	X	X	X		
LSR (1)		X	X	X	X		
ORA	X	X	X	X	X	X	X
ROL (1)		X	X	X	X		
SBC	X	X	X	X	X	X	X
STA		X	X	X	X	X	X
STX		X	X	X			
STY		X	X	X			

(1) Accumulator A can also be an operand

(2) Indexing by Y

(3) Indirect is absolute indirect and not indexed

The 8 conditional branches use relative addressing.

The 25 other instructions not in this table use implied addressing.

VITA

The author, Robert Elmore Blanton, was born October 22, 1952, in Boston, Massachusetts. He was graduated, with Honor, in June, 1975, from Northeastern University with a Bachelor of Science in Mathematics. Since September, 1976, he has been employed as a Graduate Research Assistant in the Department of Computer Science at the University of Illinois at Urbana-Champaign. He is a member of the Association for Computing Machinery.

BIOGRAPHIC DATA EET	1. Report No. UIUCDCS-R-77-886	2.	3. Recipient's Accession No.
	Title and Subtitle A CROSS-ASSEMBLER/SIMULATOR FOR A MICROPROCESSOR-BASED INDUSTRIAL CONTROLLER		5. Report Date July 1977
Author(s) Robert Elmore Blanton		6.	
Performing Organization Name and Address Department of Computer Science University of Illinois at Urbana-Champaign Urbana, Illinois 61801		8. Performing Organization Rept. No.	
Sponsoring Organization Name and Address Department of Computer Science University of Illinois at Urbana-Champaign Urbana, Illinois 61801		10. Project/Task/Work Unit No.	
		11. Contract/Grant No.	
		13. Type of Report & Period Covered M.S. Thesis	
		14.	
Supplementary Notes			
Abstracts Industrial process control is one of many areas that has benefited from the use of microprocessors. This report discusses some software tools, as implemented at the University of Illinois at Urbana-Champaign, which allow the user to effectively develop, implement and debug software for these industrial controllers. The system is also usable in many other applications utilizing the MOS Technology 650x family of microprocessors and, optionally, the MOS Technology 6520 Peripheral Interface Adapter (PIA). The software package includes a cross-assembler and simulator for the 650x family of microprocessors and, included as an integral part of the 650x simulator, a simulator for the 6520 PIA.			
Key Words and Document Analysis. 17a. Descriptors microprocessor assembler simulator industrial process control			
1b. Identifiers/Open-Ended Terms			
1c. COSATI Field/Group			
Availability Statement		19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 72
		20. Security Class (This Page) UNCLASSIFIED	22. Price

OCT 2 1980



UNIVERSITY OF ILLINOIS-URBANA
S10.84 IL6R no. C002 no.886-893(1977
Generating binary trees lexicographically



3 0112 088403594